Appendix A

```
////////////////////////////////////////////////////////////////////////
// Synchronization hub.

void Synchronize( void )
{
      GetActions( );
      ResolveConflicts( );
      PerformActions( );

      return;
}

////////////////////////////////////////////////////////////////////////
// Get the actions from the sources and GUD.

void GetActions ( )
{
      // Iterate through all of the managers and add their actions to the
list.
      for ( TSObject* pObj = m_vecSources.First();
            pObj;
              pObj = m_vecSources.Next ( ) )
      {
            TSSource* pSource = (TSSource*) pObj;

            // Get the record map from the store for this manager.
            TSSourceManager*  pManager = pSource->SourceManager ( );
            TSRecordMap*            pMap    = pManager->RecordMap ( );
            TSStore*                pStore    = pMap->Store ( );

            // Get the number of items being operated on.
            TSUINT32 uSourceCount = pSource->Count ( );
            TSUINT32 uMapCount    = pMap->MapItemCount ( );

            // Filter the gud for this specific source.
            m_pStore->Filter ( pSource );

            // Get the last synchronization time for the source itself.
            TSDateTimeStamp& tsLastSync = pMap->LastSync ( );

            // Generate the source update actions.
            int iAddCount = GetActions_SourceUpdates ( pSource, pMap,
tsLastSync );

            // Generate the source delete actions.
            GetActions_SourceDeletes ( pSource,
                                        pMap,
                                        tsLastSync,
                                        ( (long)uSourceCount
                                            - (long)uMapCount
                                            - (long)iAddCount
                                        ) != 0 );

            // Generate the GUD update actions.
            GetActions_GudUpdates ( pSource, pMap );

            // Generate the GUD delete actions.
            GetActions_GudDeletes ( pSource, pMap );
```

```
        }

        // Remove the filtering which was put in place for a given source
        m_pStore->Filter ( NULL );

        return;
}

////////////////////////////////////////////////////////////////////////////////
// Generate the source update actions.

TSINT32 GetActions_SourceUpdates (
        TSSource*               pSource,
        TSRecordMap*            pMap,
        TSDateTimeStamp&  tsLastSync
        )
{
        TSDateTimeStamp   dtsLastModification;

        // Filter the source based on the last syncrhonization time.  This
        // will ensure optimal performance for sources which can offer the
        // filter.
        pSource->Filter ( TSSOURCE_FILTER_MODIFICATIONS,
pMap->LastModification ( ) );

        // Iterate through each record in the source and determine whether
        // or not the record has been modified since the last synchronization
        TSINT32 iAddCount = 0;

        if ( pSource->MoveFirst ( ) )
        {
                do
                {
                        // Get the item to operate on.
                        TSString                strID = pSource->ID ( );
                        TSRecordMapItem*  pItem    = pMap->CurrentMapItem (
TSRECORDMAP_MAP_SOURCEID, (TSUINT32)(TSCSTR)strID );
                        TSRecordAction*         pAction = NULL;

                        TSDateTimeStamp dtsSourceMod = pSource->LastModified ( );
                        TSUINT32        uCRC         = pSource->CRC ( );

                        // If the record exists in the map then this is an update
                        // not an add.
                        if ( pItem )
                        {
                                // If there was a CRC value returned from the
source we should assume that
                                // the source does not have last modification times
on the record level and
                                // we should compare the last known crc with the
given one to determine
                                // modification.
                                if ( uCRC != 0 )
                                {
                                        if ( uCRC != pItem->CRC ( ) )
                                                pAction = new TSRecordAction (
TSRECACTIONTYPE_GUD_UPDATE, pSource, pItem );
```

```
                        }
                        else
                        {
                              if ( dt~~s~~SourceMod > pMap->LastModification (
) )
                                    pAction = new TSRecordAction (
TSRECACTIONTYPE_GUD_UPDATE, pSource, pItem );
                        }
                  }
                  // If the record did not exist in the record map it must
be a new record.
                  // Therefor we can add a new gud record and create a map
for it.
                  else
                  {
                        TSRecord* pRecord = m_pStore->CreateRecord ( );

                        pItem   = pMap->CreateMapItem ( pSource->ID ( ),
pRecord );
                        pAction = new TSRecordAction (
TSRECACTIONTYPE_GUD_ADD, pSource, pItem );

                        iAddCount++;
                  }

                  // Append the action to the list if one was created.
                  if ( pAction )
                  {
                        // Set the conflict stamp in the action.
                        pAction->ConflictStamp ( dtsSourceMod );


                        // Load the body object for this record.
                        pAction->GudRecord~~( )~~->LoadBody ( );

                        // Save a copy of the gud record and make sure it
gets written
                        // to the temporary file for the time being.
                        TSRecord* pNewRecord = (TSRecord*)
pAction->GudRecord ( )->Copy ( );
                        pNewRecord->Temporary ( TSBOOL_TRUE );

                        // Unload the body object.
                        pAction->GudRecord()->BodyObject ( NULL );

                        // Get the record from the source

                        pSource->Get ( pNewRecord );

                        // Setup the action list.
                        pAction->TempRecord ( pNewRecord );

                        pItem->SourceID ( pSource->ID ( ) );
                        pItem->CRC      ( uCRC );

                        AppendAction ( pAction );

                        // Increase the synchronization totals.
                        if ( pAction->Type ( ) == TSRECACTIONTYPE_GUD_ADD )
                              pSource->m_uAdditionsOut++;
                        else
                              pSource->m_uUpdatesOut++;

                        // If this record was modified later than any other
                        // new record we should indicate so in our last
                        // category sync time.
                        if ( dtsSourceMod > dtsLastModification && uCRC ==
0 )
                        {
                              dtsLastModification = dtsSourceMod;
                              pMap->LastRecordID ( pItem->SourceID ( ) );
                        }

                        // Save the temp record to the temporary file and
                        // clear the memory used for it.
                        pNewRecord->SaveBody ( );
                        pNewRecord->BodyObject ( NULL );
                  }
            }
            while ( pSource->MoveNext ( ) );
      }

      return iAddCount;
}

//////////////////////////////////////////////////////////////////////////
// Generate the source delete actions.

void GetActions_SourceDeletes (
      TSSource*               pSource,
      TSRecordMap*            pMap,
      TSDateTimeStamp&  dtsLastSync,
      TSBOOL                        bKnownDelete
      )
{
      // If the source responds to a filter for deletions then
      // get the deletions directly from them.
      if ( tsSuccess == pSource->Filter ( TSSOURCE_FILTER_DELETIONS,
dtsLastSync ) )
      {
            if ( tsSuccess == pSource->MoveFirst ( ) )
            {
                  do
                  {
                        // Check to see if the record told be deleted
acutally
                        // exists in our record map.
                        TSRecordMapItem* pItem = pMap->CurrentMapItem (
TSRECORDMAP_MAP_SOURCEID, (TSUINT32)(TSCSTR)pSource->ID ( ) );
                        if ( NULL == pItem )
                              continue;

                        // Create the delete action and add it to the
action vector.
                        AppendAction ( TSRECACTIONTYPE_GUD_DELETE, pSource,
pItem );
```

```
                pSource->m_uDeletionsOut++;

            } while ( tsSuccess == pSource->MoveNext ( ) );
        }
    }
    else
    {
        // Determine if there are any deletinons.  If there are find
them.
        if ( TSBOOL_FALSE == bKnownDelete )
            return;

        // Determine all of the deletions for a given source.
        if ( pMap->CurrentMapItem ( TSRECORDMAP_MAP_FIRST ) )
        {
            do
            {
                // If the record does not exist in the map, mark it
for delete
                if ( tsSuccess != pSource->MoveTo (
pMap->CurrentMapItem()->SourceID ( ) ) )
                {
                    AppendAction ( TSRECACTIONTYPE_GUD_DELETE,
                                            pSource,
pMap->CurrentMapItem ( ) );

                    pSource->m_uDeletionsOut++;
                }
            }
            while ( pMap->CurrentMapItem ( TSRECORDMAP_MAP_NEXT ) );
        }
    }

    return;
}

/////////////////////////////////////////////////////////////////////////
// Generate the GUD update actions.

void GetActions_GudUpdates (
    TSSource*           pSource,
    TSRecordMap*        pMap
    )
{
    // Tell the source to stop filtering on additions/modifications
    pSource->Filter ( TSSOURCE_FILTER_CLEAR, TSDateTimeStamp() );

    // Determine if the GUD has any record for the source.
    if ( m_pStore->CurrentRecord ( TSSTORE_RECORD_FIRST ) )
    {
        do
        {
            // Get the current record from the store.
            TSRecord* pRecord = m_pStore->CurrentRecord ( );

            // If the store item is not in the record map it
            // can be marked as an add to that source.
```

```
                    TSRecordMapItem* pItem = pMap->CurrentMapItem (
TSRECORDMAP_MAP_RECORDID, pRecord->UniqueID ( ) );
                    if ( NULL == pItem )
                    {
                          pItem = pMap->CreateMapItem ( NULL, pRecord );
                          AppendAction ( TSRECACTIONTYPE_CLIENT_ADD, pSource,
pItem );
                    }
                    // If the item exists in the GUD, check its timestamp
                    // to the Record maps timestamp for last sync. If the
                    // the GUD record is newer we have and update
                    else
                    {
                          // If the record was modified later than the last
sync time
                          // of the specific record then we should mark it as
an update.
                          if ( pRecord->LastModified ( ) > pItem->LastSync (
) )
                                AppendAction ( TSRECACTIONTYPE_CLIENT_UPDATE,
pSource, pItem );
                    }
              }
              while ( m_pStore->CurrentRecord ( TSSTORE_RECORD_NEXT ) );
        }

        return;
}

///////////////////////////////////////////////////////////////////////
// Generate the GUD delete actions.

void GetActions_GudDeletes (
      TSSource*            pSource,
      TSRecordMap*         pMap
      )
{
      // To determine whether or not there are deletions coming from the
GUD we just
      // need to find all records in the record map which have the deletion
flag set on
      if ( pMap->CurrentMapItem ( TSRECORDMAP_MAP_FIRST ) )
      {
            do
            {
                  // If the record in the gud has been deleted, we can
issue a delete
                  // to the client.
                  if ( pMap->CurrentMapItem()->Record( )->Deleted ( ) ==
true )
                        AppendAction ( TSRECACTIONTYPE_CLIENT_DELETE,
pSource,
                                          pMap->CurrentMapItem ( ) );
            }
            while ( pMap->CurrentMapItem ( TSRECORDMAP_MAP_NEXT ) );
      }

      return;
```

```
}

///////////////////////////////////////////////////////////////////////
// Resolve any action conflicts.

void ResolveConflicts ( )
{
      // Build the conflicts vector.
      BuildConflictsVector ( );

      // Resolve any conflicts which can automatically be done.
      ResolveAutomaticConflicts ( );

      // If there are still conflicts to resolve we must be using manual
      // resolution, therefore we need to allow the user to fixup the
conflicts.
      if ( m_vecConflicts.Size ( ) > 0 )
            DisplayDialog ( );

      // Purge actions. Run through them backwards so that the delete
numbers
      // stay valid as we are deleting them.
      for ( TSNumber* pnumAction = (TSNumber*)m_vecDelActions.Last();
            pnumAction;
              pnumAction = (TSNumber*)m_vecDelActions.Prev ( ) )
      {
            TSRecordAction* pAction = (TSRecordAction*)(*m_pvecActions) [
pnumAction->Value ( ) ];
            if ( pAction == NULL )
                  continue;

            // Delete action.
            pAction->TempRecord ( NULL );

            // If this type was an add then we can just delete the record
map item since
            // it isnt already in a list somewhere.
            if ( pAction->Type ( ) == TSRECACTIONTYPE_CLIENT_ADD )
                  delete pAction->RecordMapItem ( );

            m_pvecActions->Delete ( pnumAction->Value ( ) );
      }

      return;
}

///////////////////////////////////////////////////////////////////////
// Build the initial conflicts list.

void BuildConflictsVector ( )
{
      TSActionConflict* pConflict = new TSActionConflict;

      // Loop through all of the actions in the given action vector and
      // find the conflicts
      for ( TSUINT32 uAction = 0; uAction < m_pvecActions->Size(); )
      {
            TSRecordAction* pAction = (TSRecordAction*)
(*m_pvecActions)[uAction];

            TSUINT32 uRecID = pAction->GudRecord()->UniqueID ( );

            // Loop while the actions act on the same record.  If there is
more
            // than one action acting on the same record then we have a
conflict.
            do
            {
                  TSRecordAction* pAction = (TSRecordAction*)
(*m_pvecActions)[uAction];

                  if ( pAction->GudRecord ( )->UniqueID ( ) == uRecID )
                        pConflict->m_vecActions.Append ( uAction );
                  else
                        break;

                  uAction++;
            }
            while ( uAction < m_pvecActions->Size ( ) );

            // If there is more than one action acting on the current
record id
            // we have a conflict.
            if ( pConflict->m_vecActions.Size ( ) > 1 )
            {
                  m_vecConflicts.Append ( pConflict );
                  pConflict = new TSActionConflict;
            }
            else
                  pConflict->m_vecActions.Clear ( );
      }

      delete pConflict;

      return;
}

///////////////////////////////////////////////////////////////////////
// Resolve the automatic conflicts.

void ResolveAutomaticConflicts ( )
{
      TSBitField& flags = TSApplication::Config ( )->BitField (
APPCFG_GENERALFLAGS );
      TSBOOL bAutomatic = flags.Bit ( APPCFG_FLAGS_AUTOCONFLICT );

      // Iterate through all of the conflicts and resolved all which
      // can be automatically be resolved.
      for ( TSUINT32 uConflict = 0; uConflict < m_vecConflicts.Size ( );  )
      {
            TSActionConflict* pConflict =
(TSActionConflict*)m_vecConflicts[uConflict];

            TSBOOL bResolved = ResolveAutomaticConflict ( pConflict,
bAutomatic );
```

```
            // If the conflict was resolved, we can remove it from the
list.
            if ( bResolved )
                  m_vecConflicts.Delete ( uConflict );
            else
                  uConflict++;
      }

      return;
}

////////////////////////////////////////////////////////////////////////
// Resolve the conflict.

TSBOOL ResolveAutomaticConflict (
      TSActionConflict* pConflict,
      TSBOOL                        bAuto
      )
{
      TSBOOL bResolved = TSBOOL_TRUE;

      // Copy the action array;
      TSNumberVector vecActionNums;
      for ( TSNumber* pnumAction = pConflict->m_vecActions.First();
            pnumAction;
              pnumAction = pConflict->m_vecActions.Next() )
      {
            vecActionNums.Append ( pnumAction->Value ( ) );
      }

      // Step 1.  Iterate through all of the actions and resolve any
conflicts between
      //          two actions acting on the same source.
      for ( TSUINT32 uAction = 0; uAction < vecActionNums.Size(); )
      {
            // Get the first action to work on.
            TSRecordAction* pAction = (TSRecordAction*)
                  ((*m_pvecActions) [ ((TSNumber*)vecActionNums[ uAction
])->Value() ]);

            // Search forward in the action vector for actions which have
the same
            // source as the current action.
            TSBOOL bAdvance = TSBOOL_TRUE;
            for ( TSUINT32 uAction2 = uAction + 1;
                    uAction2 < vecActionNums.Size(); uAction2 ++ )
            {
                  // Get the first action to work on.
                  TSRecordAction* pAction2 = (TSRecordAction*)
                        ((*m_pvecActions) [ ((TSNumber*)vecActionNums[
uAction2 ])->Value() ]);

                  // If the two actions do not have the same source then
continue on.
                  if( pAction2->Source ( ) != pAction->Source ( ) )
                        continue;
                    if ( pAction->ConflictStamp ( ) > pAction2->ConflictStamp
( ) )
                    {
                          m_vecDelActions.Append ( ((TSNumber*)vecActionNums[
uAction2 ])->Value ( ) );
                          vecActionNums.Delete ( uAction2 );
                    }
                    else
                    {
                          m_vecDelActions.Append ( ((TSNumber*)vecActionNums[
uAction ])->Value ( ) );
                          vecActionNums.Delete ( uAction );
                          bAdvance = TSBOOL_FALSE;
                    }

                    break;
              }

              if ( bAdvance )
                    uAction++;
        }

        // Step 2/3.  Purge all client actions if there is at least one gud
action.
        TSRecordAction* pFirstAction = (TSRecordAction*)
              (*m_pvecActions)[((TSNumber*)vecActionNums[0])->Value()];

        if ( TSRECACTIONTYPE_GUD_UPDATE == pFirstAction->Type ( ) ||
             TSRECACTIONTYPE_GUD_DELETE == pFirstAction->Type ( )     )
        {
              for ( TSUINT32 uAction = 0; uAction < vecActionNums.Size(); )
              {
                    // Get the first action to work on.
                    TSRecordAction* pAction = (TSRecordAction*)
                          (*m_pvecActions) [ ((TSNumber*)vecActionNums[
uAction ])->Value() ];

                    // Once we have hit the client actions we are done with
the
                    // conflict resolution.
                    if ( TSRECACTIONTYPE_CLIENT_DELETE == pAction->Type ( )
||
                         TSRECACTIONTYPE_CLIENT_UPDATE == pAction->Type ( )
)
                    {
                          m_vecDelActions.Append ( ((TSNumber*)vecActionNums[
uAction ])->Value() );
                          vecActionNums.Delete ( uAction  );
                    }
                    else
                          uAction ++;
              }

              // Step 3.   If the first action is a gud update then we can
remove all
              //           gud deletes since the update always takes
precedence.
              if ( TSRECACTIONTYPE_GUD_UPDATE ==
((TSRecordAction*)(*m_pvecActions)[((TSNumber*)vecActionNums[0])->Value()])
->Type ( ) )
                        for ( TSUINT32 uAction = 1; uAction < vecActionNums.Size
( ); )
                        {
                                // Get the first action to work on.
                                TSRecordAction* pAction = (TSRecordAction*)
                                        (*m_pvecActions) [ ((TSNumber*)vecActionNums[
uAction ])->Value() ];

                                // If the action is a gud delete we should purge
it.
                                if ( TSRECACTIONTYPE_GUD_UPDATE != pAction->Type (
) )
                                {
                                        m_vecDelActions.Append (
((TSNumber*)vecActionNums [ uAction ])->Value() );
                                        vecActionNums.Delete ( uAction );
                                }
                                else
                                        uAction ++;
                        }

                // If the gud action is a delete then remove all other gud
                // actions which are deltes, we only need one.
                if ( TSRECACTIONTYPE_GUD_DELETE == pFirstAction->Type ( ) )
                {
                        while ( vecActionNums.Size ( ) > 1 )
                        {
                                m_vecDelActions.Append ( ((TSNumber*)vecActionNums[
1 ])->Value() );
                                vecActionNums.Delete ( 1 );
                        }
                }
                else if ( vecActionNums.Size () > 1 )
                {
                        // Find the action with the greatest modification time.
This will
                        // be the basic of our conflict merge.
                        TSUINT32 uFirstAction = 0;
                        for ( TSUINT32 uAction = 0; uAction <
vecActionNums.Size(); uAction ++ )
                        {
                                // Get the first action to work on.
                                TSRecordAction* pAction = (TSRecordAction*)
                                        (*m_pvecActions) [ ((TSNumber*)vecActionNums[
uAction  ])->Value() ];

                                if ( pAction->ConflictStamp ( ) >
pFirstAction->ConflictStamp ( ) )
                                {
                                        pFirstAction = pAction;
                                        uFirstAction = uAction;
                                }
                        }

                        vecActionNums.Delete ( uFirstAction );
                        // Set the first action.
                        pConflict->m_pResultingAction = pFirstAction;

                        // Change the type ~~to~~ a global update.
                        pFirstAction->Type ( TSRECACTIONTYPE_GLOBAL_UPDATE );

                        for ( uAction = 0; uAction < vecActionNums.Size(); )
                        {
                                // Get the first action to work on.
                                TSRecordAction* pAction = (TSRecordAction*)
                                        (*m_pvecActions) [ ((TSNumber*)vecActionNums[
uAction  ])->Value() ];

                                // Merge the records.
                                TSMergeConflictVector vecConflicts;

                                m_pAppType->SyncTypeManager()->MergeRecords (
                                                pFirstAction->TempRecord ( ),
pAction->TempRecord ( ),
                                                pFirstAction->GudRecord(),
                                                pConflict->m_vecConflicts
                                                );

                                // If we are not automatically resolving conflicts
then determine whether or not
                                // this conflict has been resolved.
                                if ( TSBOOL_FALSE == bAuto )
                                {
                                        if ( tsSuccess != tsMergeResult )
                                                bResolved = TSBOOL_FALSE;
                                        else if ( pConflict->m_vecConflicts.Size ( )
> 0 )
                                        {
                                                bResolved = TSBOOL_FALSE;
                                                m_bFieldConflict = TSBOOL_TRUE;
                                        }
                                }

                                if ( TSBOOL_TRUE == bAuto || tsSuccess ==
tsMergeResult )
                                {
                                        // Delete the unnecessary action.
                                        m_vecDelActions.Append (
((TSNumber*)vecActionNums[ uAction ])->Value() );
                                        vecActionNums.Delete ( uAction );
                                }
                                else
                                        uAction++;
                        }
                }
        }

        return bResolved;
}

///////////////////////////////////////////////////////////////////////
// Perform the actions.
```

```
void PerformActions ( )
{
      // Iterate through all of the actions in the action vector and
      // perform each.  This function assumes that any conflicts in the
      // actions are already resolved.
      for ( TSRecordAction* pAction = (TSRecordAction*) m_vecActions.First
( );
            pAction;
              pAction = (TSRecordAction*) m_vecActions.Next ( ) )
      {
            TSApplicationSource* pAppSrc =
pAction->Source()->SourceManager()->ApplicationSource( );

            PerformAction ( pAction );
      }

      return;
}

void PerformAction ( TSRecordAction* pAction )
{
      TSRecordMapItem* pItem        = pAction->RecordMapItem ( );
      TSSource*        pSource      = pAction->Source ( );
      TSRecord*          pGudRecord = pAction->GudRecord ( );
      TSRecordMap*       pMap       = pSource->SourceManager()->RecordMap (
);

      pSource->RecordMapItem ( pItem );

      switch ( pAction->Type ( ) )
      {
            case TSRECACTIONTYPE_CLIENT_ADD:
                  {
                  // Add the record to the source.
                  pSource->Add ( *pGudRecord );

                  TSString strID = pSource->ID ( );
                  pMap->CurrentMapItem ( TSRECORDMAP_MAP_SOURCEID,
(TSUINT32)(TSCSTR) strID );

                  // Save the clients crc for this record in the record
map.
                  pItem->CRC ( pSource->CRC ( ) );

                  // Fill in the source id and add the record to the map.
                  pItem->SourceID ( strID );
                  pMap->AddMapItem ( pItem );

                  // Increment the appropriate source totals.
                  pSource->m_uAdditionsIn++;

                  // Set the last sync time of the record map item to the
last
                  // modified time of the record.
                  pItem->LastSync ( pGudRecord->LastModified ( ) );

                  if ( pItem->CRC ( ) == 0 )
                        pMap->LastRecordID ( pItem->SourceID ( ) );
```

```
                    break;
                    }

            case TSRECACTIONTYPE_CLIENT_UPDATE:
                    {
                    // Move to the record which needs to be updated and
attempt to
                    // update it.
                    if ( pItem->SourceID ( ).Length ( ) == 0
||
                         tsSuccess != pSource->MoveTo ( pItem->SourceID ( )
)     )
                    {
                        pMap->RemoveMapItem ( pItem );
                        pAction->Type ( TSRECACTIONTYPE_CLIENT_ADD );
                        PerformAction ( pAction );
                        return;
                    }

                    pSource->Update ( *pGudRecord );

                    TSString strID = pSource->ID ( );
                    TSRecordMapItem* pFindItem = pMap->CurrentMapItem (
TSRECORDMAP_MAP_SOURCEID,

                    (TSUINT32)(TSCSTR) strID );

                    // Save the clients crc for this record in the record
map.
                    pItem->CRC ( pSource->CRC ( ) );

                    // Get the source ID again, in case it changed.
                    pItem->SourceID ( strID );
                    pItem->LastSync ( pGudRecord->LastModified ( ) );

                    // Increment the appropriate source totals.
                    pSource->m_uUpdatesIn++;

                    if ( pItem->CRC ( ) == 0 )
                        pMap->LastRecordID ( pItem->SourceID ( ) );

                    break;
                    }

            case TSRECACTIONTYPE_CLIENT_DELETE:
                    {
                    // Move to the item which needs to be deleted.
                    pSource->MoveTo ( pItem->SourceID ( );

                    pSource->Delete ( );

                    // Increment the appropriate source totals.
                    pSource->m_uDeletionsIn++;

                    // Delete the item from the record map.
                    pMap->DeleteMapItem ( pItem );

                    break;
```

```
                }

            case TSRECACTIONTYPE_GUD_ADD:
                // Load the body for the temporary record and prevent the
                // record from being re-written to the body file by
setting the
                // memory only flag.
                pAction->TempRecord()->LoadBody ( );
                pAction->TempRecord()->Flags ( ).Bit ( TSRECFLAG_MEMONLY,
TSBOOL_TRUE );

                // Copy the data from the record to the gud record.
                pGudRecord->CopyDataFrom ( pAction->TempRecord ( ) );

                // Get rid of the temp record
                pAction->TempRecord ( NULL );

                if ( tsDuplicate == m_pStore->AddRecord ( pGudRecord ) )
                {
                    // Add to the number of records which were merged
out.
                    m_iMergedRecords++;

                    TSRecord* pDupe = m_pStore->DuplicateRecord ( );

                    TSMergeConflictVector vecConflicts;
                    if ( tsSuccess !=
m_pAppType->SyncTypeManager()->MergeRecords (
                                    pDupe,
                                    pGudRecord,
                                    pDupe,
                                    vecConflicts ) )
                    {
                        if ( pDupe->ConflictStamp () <
pAction->ConflictStamp ( ) )
                        {
                            pDupe->LoadBody ( );
                            pDupe->CopyDataFrom ( pGudRecord );
                            pDupe->ConflictStamp (
pAction->ConflictStamp ( ) );
                            pDupe->LastModified (
TSDateTimeStamp::CurrentTime ( ) );

                            UpdateAllSources ( pDupe );
                        }
                    }
                    else
                    {
                        if ( pAction->ConflictStamp ( ) >
pDupe->ConflictStamp ( ) )
                            pDupe->ConflictStamp (
pAction->ConflictStamp ( ) );
                        pDupe->LastModified (
TSDateTimeStamp::CurrentTime ( ) );
                        UpdateAllSources ( pDupe );
                    }
```

```
                        pDupe->SaveBody ( );
                        pDupe->BodyObject ( NULL );

                        // Delete the record which was found to be a
duplicate.
                        if ( tsSuccess == pSource->MoveTo ( pItem->SourceID
( ) ) )
                        {
                              pSource->Delete ( );
                              m_vecTrashCan.Append ( pItem );
                              m_vecTrashCan.Append ( pGudRecord );
                        }
                  }
                  else
                  {
                        pMap->AddMapItem ( pItem );
                        pItem->LastSync ( pGudRecord->LastModified ( ) );

                        // Set the conflict stamp for this record.
                        pGudRecord->ConflictStamp ( pAction->ConflictStamp
( ) );


                        ExpandGudAction ( pAction );
                  }

                  // Ensure the body of the gud record is no longer loaded.
                  pGudRecord->BodyObject( NULL );

                  break;

            case TSRECACTIONTYPE_GLOBAL_UPDATE:
            case TSRECACTIONTYPE_GUD_UPDATE:
                  {
                  // Load the body for the temporary record and prevent the
                  // record from being re-written to the body file by
setting the
                  // memory only flag.
                  pAction->TempRecord()->LoadBody ( );
                  pAction->TempRecord()->Flags ( ).Bit ( TSRECFLAG_MEMONLY,
TSBOOL_TRUE );

                  // Copy the data from the record to the gud record.
                  pGudRecord->CopyDataFrom ( pAction->TempRecord ( ) );

                  // Get rid of the temp record
                  pAction->TempRecord ( NULL );

                  if ( TSRECACTIONTYPE_GLOBAL_UPDATE != pAction->Type ( ) )
                        pItem->LastSync ( pGudRecord->LastModified ( ) );

                  // Set the conflict stamp for this record.
                  pGudRecord->ConflictStamp ( pAction->ConflictStamp ( ) );

                  ExpandGudAction ( pAction );

                  // Unload the body object
                  pGudRecord->SaveBody ( );
                  pGudRecord->BodyObject ( NULL );
```

```
                    break;
                    }

                case TSRECACTIONTYPE_GUD_DELETE:

                    // Mark the GUD record as deleted.
                    pGudRecord->Deleted ( TSBOOL_TRUE );
                    pGudRecord->LastModified ( TSDateTimeStamp::CurrentTime (
) );

                    // Set the conflict stamp for this record.
                    pGudRecord->ConflictStamp ( pAction->ConflictStamp ( ) );

                    ExpandGudAction ( pAction );

                    // Remove the item which caused the delete to occurr.
                    pMap->DeleteMapItem ( pItem );

                    break;
        }
}

void ExpandGudAction (
        TSRecordAction* pAction
        )
{
        TSRECORDACTIONTYPE eType;

        // convert the original record action type to the
        // expanded type.
        switch ( pAction->Type ( ) )
        {
                case TSRECACTIONTYPE_GUD_ADD:
                        eType = TSRECACTIONTYPE_CLIENT_ADD;
                        break;

                case TSRECACTIONTYPE_GUD_UPDATE:
                case TSRECACTIONTYPE_GLOBAL_UPDATE:
                        eType = TSRECACTIONTYPE_CLIENT_UPDATE;
                        break;

                case TSRECACTIONTYPE_GUD_DELETE:
                        eType = TSRECACTIONTYPE_CLIENT_DELETE;
                        break;
        }

        // Extract the gud record to use in the following loop
        TSRecord* pGudRecord = pAction->GudRecord ( );

        // Issue the delete to all other clients involved in the
        // synchronization.
        for ( TSSource* pSource = (TSSource*) m_vecSources.First ( );
                pSource;
                pSource = (TSSource*) m_vecSources.Next ( ) )
        {
                // Dont perform any actions to this source if it is full.
                TSApplicationSource* pAppSrc =
pSource->SourceManager()->ApplicationSource ( );
```

```
                if ( pAppSrc->Flags ( ).Bit ( SOURCE_FLAG_LOWMEMORY ) )
                     continue;

                if ( pSource == pAction->Source ( )                    &&
                    TSRECACTIONTYPE_GLOBAL_UPDATE != pAction->Type ( )      )
                     continue;

                // If this record does not belong on the current source we
                // should no consider it.
                if ( TSBOOL_TRUE == FilterSourceRecord ( pSource, pGudRecord )
)
                     continue;

                TSRecordMap*     pMap   = pSource->SourceManager ( )->RecordMap
( );
                TSRecordMapItem* pItem = pMap->CurrentMapItem (
TSRECORDMAP_MAP_RECORDID, pGudRecord->UniqueID ( ) );

                if ( NULL == pItem )
                {
                     // If the item is NULL and the action is a delete action,
it
                     // means the record is not in the source so we dont have
                     // to delete it.
                     if ( eType == TSRECACTIONTYPE_GUD_DELETE )
                          continue;

                     // Create a new map to use in the perform function.  This
should
                     // happen always if the type is ADD and could possibly
happend
                     // if the type is UPDATE and the record does not yet
exist on the
                     // destinate source.
                     pItem = pMap->CreateMapItem ( NULL, pGudRecord );
                }

                // Perform the expanded action.
                PerformAction ( &TSRecordAction ( eType,pSource,pItem ) );
        }

        return;
}

void UpdateAllSources ( TSRecord* pGudRecord )
{
        // Loop through all of the sources.
        TSRecordAction Action;
        for ( TSUINT32 uSource = 0; uSource < m_vecSources.Size(); uSource++
)
        {
                TSSource*                  pSource = (TSSource*) m_vecSources [
uSource ];
                TSRecordMap*               pMap =
pSource->SourceManager()->RecordMap ( );
                TSRecordMapItem*  pItem = pMap->CurrentMapItem (
TSRECORDMAP_MAP_RECORDID, pGudRecord->UniqueID ( ) );
```

```
            if ( NULL == pItem )
                  continue;

            // Build the action
            Action.RecordMapItem ( pItem );
            Action.TempRecord( NULL );
            Action.Source ( pSource );
            Action.Type ( TSRECACTIONTYPE_CLIENT_UPDATE );

            // Now perform the action.
            PerformAction ( &Action );
      }

      return;
}
```